Lazy Lexing is Fast

Manuel M. T. Chakravarty

Institute of Information Sciences and Electronics
University of Tsukuba, Japan
`chak@is.tsukuba.ac.jp`
`www.score.is.tsukuba.ac.jp/~chak/`

Abstract This paper introduces a set of combinators for building lexical analysers in a lazy functional language. During lexical analysis, the combinators generate a deterministic, table-driven analyser on the fly. Consequently, the presented method combines the efficiency of off-line scanner generators with the flexibility of the combinator approach. The method makes essential use of the lazy semantics of the implementation language Haskell. Finally, the paper discusses benchmarks of a scanner for the programming language C.

1 Introduction

There are two conceptually different approaches to obtaining a functional implementation of a scanner or parser from a formal lexical or syntactic specification: (1) the specification is written in a special purpose language and translated into a functional program by a scanner or parser generator or (2) the specification is composed from a set of combinators provided by a scanner or parser combinator library.

Both approaches have their advantages and were pursued in the past. There are a couple of generators (e.g., [8,2,5]) for widely used languages, such as Haskell and SML, as well as a number of combinator libraries (e.g., [3,4]). Usually, generators produce more efficient scanners and parsers by implementing table-driven analysers [1], but this is, in the case of parsers, at the expense of expressiveness, as generators usually confine the specification to a restricted class of languages, such as, LALR(1). Furthermore, combinator libraries are easier to handle—no separate input language and generator program are necessary—and they can specify analysers for statically unknown grammars, i.e., the grammar can change during program execution. Recently, Swierstra and Duponcheel [7] introduced self-optimising parser combinators, which during parsing generate a deterministic analyser for LL(1) grammars, thus, combining the flexibility of combinators with the efficiency of generators.

The present paper introduces a related technique for lexical analysers (*lexers*, for short). During application of the lexer, a deterministic, table-driven analyser is generated on the fly from a combinator-based lexical specification. Tokens are produced by user-defined *actions*, the standard *principle of the longest match* is applied [1], and non-regular features, such as nested comments, are supported by *meta actions.* The described technique is fully implemented by a Haskell library, called `Lexers`, and the code is available for public use.[1] The efficiency of the technique was evaluated by

---

[1] Download at `http://www.score.is.tsukuba.ac.jp/~chak/ctk/`

benchmarking an implementation of a lexer for the programming language C. Despite the relation to the mentioned work of Swierstra and Duponcheel on parser generators, the presented technique is substantially different from theirs.

An interesting aside is the use made of Haskell's lazy evaluation semantics. The state transition tables generated by the lexer combinators to achieve deterministic behaviour are represented by a cyclic tree structure, which requires either non-strict semantics or impure mutable structures. As an additional advantage of laziness, only those fragments of the table are constructed that are actually needed to accept the input at hand. Thus, we avoid overly high startup costs for short inputs, which would be incurred in non-strict, but eager (= lenient) languages like pH [6].

In summary, the central contributions are the following:

- A new technique for on-line generation of state transition tables from regular expressions in a lazy functional language.
- A combinator library for specifying lexical analysers, including often required non-regular features.
- Experimental evaluation of the proposed table generation technique.

The paper is structured as follows. Section 2 introduces the basic ideas of the combinator library and table generation. Section 3 defines the interface of the combinator library and Section 4 formalises the dynamic table generation. Section 5 presents benchmarks. Finally, Section 6 discusses related work and Section 7 concludes.

2 Combinators and Automata

Specifications of lexers essentially consist of pairs of *regular expressions* and *actions*, where the latter are program fragments that are executed as soon as the associated regular expression matches the current input. Actions usually generate tokens, which are consumed by a parser following the lexer. Regular expressions typically have the following structure [1]:

- ϵ matches the empty word.
- A single character $\mathtt{c}$ matches itself.
- $r_1 r_2$ matches a word of which the first part is matched by r_1 and the second part by r_2.
- $r_1 \mid r_2$ matches a word that is either matched by r_1 or by r_2.
- $r*$ matches a word that consists of zero, one, or more successive words matched by r.
- $r+$ matches a word that consists of one or more successive words matched by r.
- $r?$ matches a word that is either empty of matched by r.

Furthermore, we use $[c_1 \ldots c_n]$ to abbreviate $c_1 \mid \cdots \mid c_n$. For example, integer numbers are matched by $-?[0 \ldots 9]+$, i.e., an optional minus sign is followed by a non-empty sequence of digits. The same regular expression can be expressed in Haskell as follows (we define the used combinators in detail in the next section):

```
(char '-')`quest` (alt ['0'..'9'])`plus` epsilon
```

or, in this paper with improved typesetting, as $(\mathit{char}\ \text{'}-\text{'})\ \textsf{?}\ (\mathit{alt}\,[\text{'0'..'9'}]) \oplus \epsilon$. In a lexer specification, this regular expression would be paired with an action that, given the string—or, *lexeme*—of a recognised integer, produces the corresponding token.

A well known result from automata theory states that, for each regular expression, there exists a *finite state automaton (FA)* that accepts exactly those words that are matched by the regular expression; more precisely, there even exists a *deterministic finite state automaton (DFA)*, i.e., one where in each state, by looking at the next input symbol, we can decide deterministically into which state the automaton has to go next. This correspondence is exploited in scanner generators to produce a deterministic, table-driven analyser from a lexical specification. The technique presented here builds on the same theory. An additional complication, when handling a full lexical specification, as opposed to a single regular expression, is the requirement to match against a set of regular expressions in a single pass.

By implementing matching of regular expression by a DFA, we get a *table driven lexer*. Such a lexer represents the *state transition graph* of the automaton by a two-dimensional table. During lexical analysis, it traverses the transition graph by repeatedly indexing the table with the current automaton state and input symbol to obtain the next state. In the standard approach, the table is constructed off-line by a scanner generator. More precisely, the regular expressions are either first transformed into a *nondeterministic finite automaton* (NFA), and then, the NFA is converted to an DFA or the DFA is directly obtained from the regular expressions; finally, the number of states of the DFA is minimised and often the table is stored in a compressed format, as it tends to be sparse. This standard technology is described in detail in compiler textbooks [9,1].

As we generate the automaton incrementally during lexing, we cannot use a multi-staged approach and we have to avoid expensive techniques, such as, the subset construction. We essentially build the state transition graph of a DFA directly from the regular expressions and we construct a node of the graph only if we actually reach it during lexing. Representing the graph directly, instead of encoding it in a table, facilitates incremental construction and reduces the storage requirements in comparison to an uncompressed table (the transition table is usually sparse).

3 The Interface

Before we discuss the details of automata construction, we have to fix the combinator set used for the lexical specification. It consists of regular expressions, actions, and meta actions, where regular expressions describe the structure of lexemes, actions specify the transformation of lexemes into tokens, and meta actions keep track of line numbers and implement non-regular features, such as, nested comments.

3.1 Regular Expressions

Sets of lexemes that should trigger the same action (i.e., which usually generate the same sort of token) are specified by regular expressions that are freely constructed from a set of combinators, which resemble the operators on regular expressions discussed in the previous section.

A regular expression is denoted by a Haskell expression of type $(\mathit{Regexp}\ s\ t)$. Such a regular expression is, according to its type arguments, used in a lexer, which maintains an internal state of type s and produces tokens of type t—it would be possible to hide the type variables s and t add this point by using the non-standard feature of explicit universal quantification and at the cost of slightly complicating some other definitions. The following combinators[2] are available for constructing regular expressions:

$\epsilon \ :: \mathit{Regexp}\ s\ t$ — empty word
$\mathit{char} :: \mathit{Char} \to \mathit{Regexp}\ s\ t$ — single character
$(\triangleright) :: \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t$ — concatenation
$(\triangleright\!\!\triangleleft) :: \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t$ — alternatives
$(\circledast) :: \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t$ — zero, one, or more rep.
$(\oplus) :: \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t$ — one or more repetitions
$(?) :: \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t \to \mathit{Regexp}\ s\ t$ — zero or one occurences

We can define $(\circledast)$, $(\oplus)$, and $(?)$ in terms of the first four combinators; we shall return to the details of these definitions later. Furthermore, the following two definitions are useful for making regular expressions more concise:

$$\begin{array}{ll}
\mathit{alt} & :: [\mathit{Char}] \to \mathit{Regexp}\ s\ t \\
\mathit{alt}\,[c_1, \ldots, c_n] & = \mathit{char}\ c_1 \triangleright\!\!\triangleleft \cdots \triangleright\!\!\triangleleft \mathit{char}\ c_n \\
 & \\
\mathit{string} & :: \mathit{String} \to \mathit{Regexp}\ s\ t \\
\mathit{string}\,[c_1, \ldots, c_n] & = \mathit{char}\ c_1 \triangleright \cdots \triangleright \mathit{char}\ c_n
\end{array}$$

Finally, the following precedence declarations apply:

$\mathit{infixl}\ 4\ \circledast,\ \oplus,\ ?$
$\mathit{infixl}\ 3\ \triangleright$
$\mathit{infixl}\ 2\ \triangleright\!\!\triangleleft$

Given these combinators, it is straight forward to define familiar token types, e.g., identifiers:

$$\begin{array}{l}
\mathit{ident} :: \mathit{Regexp}\ s\ t \\
\mathit{ident} = \mathit{letter} \triangleright (\mathit{letter} \triangleright\!\!\triangleleft \mathit{digit}) \circledast \epsilon \\
\quad \mathbf{where} \\
\qquad \mathit{letter} = \mathit{alt}\,([\text{'a'..'z'}] +\!\!\!+ [\text{'A'..'Z'}]) \\
\qquad \mathit{digit}\ \ = \mathit{alt}\,[\text{'0'..'9'}]
\end{array}$$

In other words, an identifier is composed out of a letter, followed by zero or more letters or digits. Note the use of ϵ at the end of the regular expression; $\circledast$ is an infix operator and requires a second argument. From the point of view of regular expressions, the more natural choice for $\circledast$, $\oplus$, and $?$ would be the use of (unary) postfix operators. However, Haskell does not have postfix operators and, even if it had, a unary operator would make the implementation more difficult; as we will see in Section 4, the binary nature of these operators allows an important optimization.

---

[2] This presentation takes the freedom to improve on a purely ASCII-based typesetting of Haskell expressions. In ASCII, ϵ is represented by `epsilon`, $(\triangleright)$ by `(+>)`, $(\triangleright\!\!\triangleleft)$ by `(>|<)`, $(\circledast)$ by `star`, $(\oplus)$ by `plus`, and $(?)$ by `quest`.

3.2 Actions

Whenever the lexer recognises a lexeme matching a given regular expression, it applies an action function to the lexeme and its position to produce a token.

```
type Action t = String → Position → Maybe t
```

An action may choose to produce no token (i.e., return *Nothing*); for example, if the lexeme represents white space or a comment. Using the function

```
lexaction :: Regexp s t → Action t → Lexer s t
```

we can bundle a regular expression with an action to form a lexer—the latter are denoted by types *Lexer s t* with user-defined state *s* and tokens *t*. We can disjunctively combine a collection of such lexers with the following combinator:

```
infixl 2 >||<
(>||<) :: Lexer s t → Lexer s t → Lexer s t
```

Given these functions, let us continue the above example of lexing identifiers, i.e., the definition of the regular expression *ident*. If tokens are defined as

```
data Token = IdentTok String Position
           | ···
```

we can define a lexer that recognises identifiers only by

```
lexident :: Lexer s Token
lexident = ident `lexaction` (λstr pos → Just (IdentTok str pos))
```

A compound lexer that recognises sequences of identifiers separated by space characters can be defined as follows:

```
lexer :: Lexer s Token
lexer =     lexident
      >||< char ' ' `lexaction` (λ_ _ → Nothing)
```

3.3 Meta Actions

In addition to actions that produce tokens, we have meta actions, which alter the internal behaviour of the produced lexer. Like a normal action, a meta action is a function:

```
type Meta s t = Position → s → (Position, s, Maybe (Lexer s t))
```

Given the current position and a user-defined lexer state (of type *s*), a meta action produces an updated position and an updated user-defined state. Furthermore, it may choose to return a lexer that is used, instead of the current one, to analyse the next token. The user-defined state can, for example, be used to implement a *gensym* routine or to maintain an identifier table during lexical analysis. In combination with the capability of returning a lexer, it can be used to implement non-regular features, such as, nested comments. We can realise the latter as follows:

- The user-defined state keeps track of the current nesting level.
- We use two different lexers: (a) the *standard lexer* recognises tokens outside of comments and (b) the *comment lexer* keeps track of the nesting level while scanning though comments.
- When the lexing process encounters a lexeme starting a comment, it invokes a meta action that increases the nesting count and returns (in the third component of its result) the comment lexer.
- When the lexing process encounters a lexeme ending a comment, it triggers a meta action that decreases the nesting count and returns the standard lexer if the nesting count reached zero; otherwise, it returns the comment lexer.

Note that this technique is a kind of cleaned up, side-effect free variant of the user-defined start states found in parser generators like `lex`.

Like standard actions, we combine meta actions with regular expressions into lexers:

```
lexmeta :: Regexp s t → Meta s t → Lexer s t
```

The library `Lexers` discussed here internally uses meta actions to keep track of positions in the presence of control characters in the following way (*Position* is a triple composed from a file name, row, and column):

```
ctrlLexer :: Lexer s t
ctrlLexer =
      char '\n' `lexmeta` newline
  >||< char '\r' `lexmeta` newline
  >||< char '\f' `lexmeta` formfeed
  >||< char '\t' `lexmeta` tab
  where
    newline  (fname, row, _  ) s = ((fname, row + 1, 1), s, Nothing)
    formfeed (fname, row, col) s = ((fname, row, col + 1), s, Nothing)
    tab      (fname, row, col) s = ((fname, row, col + 8 − col `mod` 8),
                                    s, Nothing)
```

3.4 Lexing

While performing lexical analysis, a lexer maintains a lexer state

```
type LexerState s = (String, Position, s)
```

which is composed out of the remaining input string, the current source position, and a user-defined component s—the latter is the same state that is transformed in meta actions. Given a lexer and a lexer state, the following function executes the lexical analysis and yields a token string:[3]

---

[3] In the actual implementation of the `Lexers` library, the function also returns the final lexer state as well as a list of error messages.

```
execLexer                :: Lexer s t → LexerState s → [t]
execLexer l ([], _, _)   = []
execLexer l state        = case lexOne l state of
                              (Nothing, l′, state′) → execLexer l′ state′
                              (Just t  , l′, state′) → t : execLexer l′ state′
```

The definition uses *lexOne* to read a single lexeme from the input stream. If a token is produced, it is added to the resulting token stream. In any case, *execLexer* recurses to process the reminder of the input—in the case, where the just read token triggered a meta action, the lexer l' used in the recursion may differ from l. The signature of *lexOne*, which will be defined in detail later, is

```
lexOne :: Lexer s t → LexerState s → (Maybe t, Lexer s t, LexerState s)
```

4 Table Construction and Lexical Analysis

Given the interface of the previous section, it is not difficult to imagine a standard combinator-based implementation (similar to [3]). The disadvantage of such a naive implementation is, however, its low efficiency (see Section 5). As mentioned, the standard method for making lexers more efficient is to transform the specification into the state transition table of a deterministic finite automaton.

The main contribution of the present paper is a purely functional algorithm that on-the-fly constructs a DFA in a compressed table representation from regular expressions. To improve the execution time for short inputs on big lexical specifications, only those parts of the table that are needed for lexing the input at hand are constructed. We achieve this by implementing the combinators building regular expressions as *smart* constructors of the transition table (instead of regarding a regular expression as a data structure, which is processed by a table construction function). More precisely, an expression of type (*Lexer s t*), which is a combination of regular expressions and actions, evaluates to a state transition table of a DFA that accepts exactly the lexemes specified by the regular expressions and has the actions in the automaton's final states—the table construction algorithm is encoded in the combinators forming regular expressions.

4.1 Table Representation

In imperative algorithms the state transition table may be represented by a two-dimensional array indexed by the current automaton state and the current input symbol. For us, such a representation, however, has two problems: (1) The table tends to be sparse and (2) we like to incrementally refine the table. The sparseness of the table is, of course, also an interesting point in imperative lexers; therefore, the table is often stored in a compressed format after the off-line table construction is completed. However, this approach is not attractive in an on-line approach (i.e., when the table is constructed during lexing). Regarding the second point, our incremental table construction requires frequent updates of the transition table (until, eventually, the table for the whole lexical specification is completed), an operation that is expensive on big arrays in a purely functional programming style.

Using the advanced data structures available in a functional language, we directly represent the state transition graph of the DFA, instead of using a conventional table representation. The state transition graph is a directed, cyclic graph—thus, a direct representation requires either a non-strict functional language or impure operations on mutable structures. Each node in the graph represents a state of the DFA and each edge represents a state transition labelled with the input character triggering the transition. As we represent a DFA (not an NFA), there is at most one outgoing edge for each possible input character from each node. The final states of the DFA are those that are associated with an action and the initial state is the graph node that we use as the root. Overall, we represent a DFA (or lexer) by the following structure:

$$\begin{array}{lcl} \mathit{data}\ \mathit{Lexer}\ s\ t & = & \mathit{State}\ (\mathit{LexAction}\ s\ t)\ [(\mathit{Char},\ \mathit{Lexer}\ s\ t)] \\ \mathit{data}\ \mathit{LexAction}\ s\ t & = & \mathit{Action}\ (\mathit{Action}\ t) \\ & \mid & \mathit{Meta}\ (\mathit{Meta}\ s\ t) \\ & \mid & \mathit{NoAction} \end{array}$$

A $\mathit{State}\ a\ c$ where a is different from $\mathit{NoAction}$ represents a final state. The continuation c is an association list of admissible transitions—i.e., for each acceptable character in the current state, it contains the node in the graph that is reached after the associated character is read. For example,

$$\mathit{State}\ \mathit{ac}\ [(\text{'a'},\ l_1),\ (\text{'c'},\ l_2),\ (\text{'d'},\ l_3)]$$

is a graph node that represents a state with three outgoing edges. On reading 'a', 'c', and 'd' the states l_1, l_2, l_3, respectively, are reached. The action ac is executed before the choice between 'a', 'c', and 'd' is made.

The actual implementation of the library `Lexers` uses two different kinds of nodes (instead of only State) to represent states: One kind is used for states with only a few outgoing edges and the other for states with many outgoing edges. The former uses an association list, like in the above definition, but the latter uses an array to associate input characters with successor nodes. This increases the efficiency of the lookup in case of many outgoing edges, but does not alter the essence of the actual table construction algorithm.

4.2 Basic Table Construction

Before discussing the table-construction operators, we have to fix the type definition of regular expressions.

$$\mathit{type}\ \mathit{Regexp}\ s\ t\ =\ \mathit{Lexer}\ s\ t\ \rightarrow\ \mathit{Lexer}\ s\ t$$

The essential point is that a regular expression is a lexer that can still be extended by another lexer to form a new lexer. In other words, if we have $re\ l$, then, if a word matching the regular expression re is accepted, the rest of the input is handed to the lexer l (which may be a final state associated with an action). In other words, we have to match an arbitrary prefix re before the automaton reaches l (remember that expressions of type $(\mathit{Lexer}\ s\ t)$ represent states of the overall automaton), whereas an individual node of the state transition graph specifies the consumption of only a single character, together with the associated transition.

Given this, the definition of ϵ, char, and $\triangleright$ is straight forward.

$$\begin{array}{lcl} \epsilon & :: & \mathit{Regexp}\ s\ t \\ \epsilon & = & \mathit{id} \\ & & \\ \mathit{char} & :: & \mathit{Char} \rightarrow \mathit{Regexp}\ s\ t \\ \mathit{char}\ c & = & \lambda l \rightarrow \mathit{State}\ \mathit{NoAction}\ [(c,\ l)] \\ & & \\ (\triangleright) & :: & \mathit{Regexp}\ s\ t \rightarrow \mathit{Regexp}\ s\ t \rightarrow \mathit{Regexp}\ s\ t \\ (\triangleright) & = & (\circ) \end{array}$$

The functions id and $(\circ)$ are the identity function and function composition, respectively. Thus, it immediately follows that ϵ is the left and right neutral of $\triangleright$, and furthermore, that $\triangleright$ is associative. Properties required by formal language theory.

The definition of $\mathit{lexaction}$ requires a little more care:

$$\begin{array}{lcl} \mathit{lexaction} & :: & \mathit{Regexp}\ s\ t \rightarrow \mathit{Action}\ t \rightarrow \mathit{Lexer}\ s\ t \\ \mathit{lexaction}\ \mathit{re}\ a & = & \mathit{re}\ (\mathit{State}\ (\mathit{Action}\ a)\ []) \end{array}$$

It adds a final state containing the given action a and no transitions to the regular expression.

The most interesting combinators are alternatives, because they introduce indeterminism when constructing an NFA; thus, as we directly construct a DFA, we immediately have to resolve this potential indeterminism. To avoid dealing with the same problems in the two combinators $\triangleright\!|\!\triangleleft$ and $\triangleright\!\|\!\triangleleft$, we define $\triangleright\!|\!\triangleleft$ in terms of $\triangleright\!\|\!\triangleleft$:

$$\begin{array}{lcl} (\triangleright\!|\!\triangleleft) & :: & \mathit{Regexp}\ s\ t \rightarrow \mathit{Regexp}\ s\ t \rightarrow \mathit{Regexp}\ s\ t \\ \mathit{re} \mathbin{\triangleright\!|\!\triangleleft} \mathit{re}' & = & \lambda l \rightarrow \mathit{re}\ l \mathbin{\triangleright\!\|\!\triangleleft} \mathit{re}'\ l \end{array}$$

This essentially means that, given a regular expression $(\mathit{re}_1 \mathbin{\triangleright\!|\!\triangleleft} \mathit{re}_2) \triangleright \mathit{re}_3$, the continuation of re_1 and re_2 is the same, namely re_3. In other words, in case of an alternative, independent of which branch we take, we reach the same suffix. Given this definition, $\triangleright\!|\!\triangleleft$ is associative and commutative if $\triangleright\!\|\!\triangleleft$ is. The latter we define as

$$\begin{array}{lcl} (\triangleright\!\|\!\triangleleft) & :: & \mathit{Lexer}\ s\ t \rightarrow \mathit{Lexer}\ s\ t \rightarrow \mathit{Lexer}\ s\ t \\ (\mathit{State}\ a\ c) \mathbin{\triangleright\!\|\!\triangleleft} (\mathit{State}\ a'\ c') & = & \mathit{State}\ (\mathit{joinActions}\ a\ a')\ (\mathit{accum}\ (\triangleright\!\|\!\triangleleft)\ (c \mathbin{+\!\!+} c')) \end{array}$$

The function combines transitions and actions separately. The two transition tables c and c' are combined by $(\mathit{accum}\ (\triangleright\!\|\!\triangleleft)\ (c \mathbin{+\!\!+} c'))$, which concatenates the association lists c and c' and applies the auxiliary functions accum to replace all pairs of conflicting occurrences $(c,\ l)$ and $(c,\ l')$ by a new pair $(c,\ l \mathbin{\triangleright\!\|\!\triangleleft} l')$ (we omit the precise definition of accum; it is slightly tedious, but does not present new insights). Finally, the following function combines two actions:

$$\begin{array}{llll}
\mathit{joinActions} :: \mathit{LexAction}\ s\ t \to \mathit{LexAction}\ s\ t \to \mathit{LexAction}\ s\ t \\
\mathit{joinActions}\ \mathit{NoAction}\ a' & & = & a' \\
\mathit{joinActions}\ a & \mathit{NoAction} & = & a \\
\mathit{joinActions}\ \_ & \_ & = & \mathit{error}\ \text{"Lexers : Ambiguous action!"}
\end{array}$$

If both of the two states have an action, *joinActions* raises an error; it implies that there is a lexeme that is associated with two actions, but we can only execute one of them. This indicates an error in the lexical specification. From the definition of *joinActions* and the associativity and commutativity of $+\!\!+$, we can easily deduce the associativity and commutativity of $\triangleright\!\!\|\!\!\triangleleft$, and thus, $\triangleright\!\!|\!\!\triangleleft$.

As an example for the use of $\triangleright\!\!\|\!\!\triangleleft$ consider

$$(\mathit{char}\ \text{'a'}\ \text{`}\mathit{lexaxtion}\text{`}\ \ ac_1)\ \triangleright\!\!\|\!\!\triangleleft\ (\mathit{char}\ \text{'a'}\ \triangleright\ \mathit{char}\ \text{'b'}\ \text{`}\mathit{lexaxtion}\text{`}\ \ ac_2)$$

The two regular expression have a common prefix; thus, the initial state of the resulting lexer only has a single transition for the character `a`. The following state contains the action ac_1—making it a final state—but also contains a transition for `b`. The third state contains only the action ac_2, no further transition is possible. Overall, we get the lexer

$$\mathit{State}\ \mathit{NoAction}\ [('a',\ \mathit{State}\ ac_1\ [('b',\ \mathit{State}\ ac_2\ [])])]$$

This example illustrates how an action can get in the middle of a more complex lexer. In this example, it may appear as if the choice between ac_1 and ac_2 were indeterministic when accepting `ab`; however, the principle of the longest match, which we shall discuss in Section 4.4, disambiguates the choice and requires the lexer to execute only ac_2.

At this point, the reason for a design decision made earlier in the definition of the *Lexer* data type also becomes clear. When considering a single regular expression and action, it might seem reasonable to pair the action with the root of the transition graph; instead of allowing an action in every state and putting the initial action at the tips of the structure representing the lexer. As we just saw, when two lexers are disjunctively combined, actions may may move in the middle of the transition graph and the exact location of an action within the graph becomes important—it is not sufficient to collect all actions at the root of the graph.

4.3 Cyclic Graphs

The remaining combinators for regular expressions, $\circledast$, $\oplus$, and $?$, are build from the basic combinators that we just discussed. The definition of $?$ is straight forward:

$$\begin{array}{lll}
(?) & :: & \mathit{Regexp}\ s\ t\ \to\ \mathit{Regexp}\ s\ t\ \to\ \mathit{Regexp}\ s\ t \\
re1\ ?\ re2 & = & (re1\ \triangleright\ re2)\ \triangleright\!\!|\!\!\triangleleft\ re2
\end{array}$$

The recursive behaviour of $\circledast$ and $\oplus$, however, requires some additional thought. In a first attempt, we might define $\circledast$ as follows:

$$\begin{array}{lll}
(\circledast) & :: & \mathit{Regexp}\ s\ t\ \to\ \mathit{Regexp}\ s\ t\ \to\ \mathit{Regexp}\ s\ t \\
re1 \circledast\ re2 & = & \mathbf{let}\ \mathit{self}\ =\ (re1\ \triangleright\ \mathit{self}\ \triangleright\!\!\|\!\!\triangleleft\ \epsilon)\ \mathbf{in}\ \mathit{self}\ \triangleright\ re2
\end{array}$$

The recursion is realised by the local definition of self, which can either be the empty word ϵ, or it can be a word matched by $re1$ and followed again by self, until finally, self is followed by $re2$. In other words, we can use $re1$ zero, one, or more times before continuing with $re2$. So, the above definition is correct, but unfortunately, it is operationally awkward. It does not create a cyclic graph, but produces an infinite path, repeating $re1$. As the path is constructed lazily, the definition works, but, for the construction of the state transition graph, it consumes memory and time proportional to the length of the accepted lexeme, rather than the size of the regular expression.

Fortunately, we can turn this inefficient definition by equational reasoning into an efficient, but harder to understand definition:

$$
\begin{array}{ll}
 & \textbf{let}\ \mathit{self}\ =\ (re1\ \rhd\ \mathit{self}\ \mathbin{\rhd\!\!\!|\!\!\lhd}\ \epsilon)\ \textbf{in}\ \mathit{self}\ \rhd\ re2 \\
= & \{\text{unfold}\ (\mathbin{\rhd\!\!\!|\!\!\lhd})\} \\
 & \textbf{let}\ \mathit{self}\ l\ =\ (re1\ \rhd\ \mathit{self})\ l\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ \epsilon\ l\ \textbf{in}\ \mathit{self}\ \rhd\ re2 \\
= & \{\text{unfold}\ \epsilon\ \text{and beta reduction}\} \\
 & \textbf{let}\ \mathit{self}\ l\ =\ (re1\ \rhd\ \mathit{self})\ l\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ l\ \textbf{in}\ \mathit{self}\ \rhd\ re2 \\
= & \{\text{unfold}\ (\rhd)\ \text{and beta reduction}\} \\
 & \textbf{let}\ \mathit{self}\ l\ =\ re1\ (\mathit{self}\ l)\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ l\ \textbf{in}\ \mathit{self}\ \rhd\ re2 \\
= & \{\text{unfold}\ (\rhd)\} \\
 & \textbf{let}\ \mathit{self}\ l\ =\ re1\ (\mathit{self}\ l)\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ l\ \textbf{in}\ \lambda l'\ \rightarrow\ \mathit{self}\ (re2\ l') \\
= & \{\text{float let inside lambda}\} \\
 & \lambda l'\ \rightarrow\ \textbf{let}\ \mathit{self}\ l\ =\ re1\ (\mathit{self}\ l)\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ l\ \textbf{in}\ \mathit{self}\ (re2\ l') \\
= & \{\text{lambda dropping}\} \\
 & \lambda l'\ \rightarrow\ \textbf{let}\ l\ =\ re2\ l';\ \mathit{self}\ =\ re1\ \mathit{self}\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ l\ \textbf{in}\ \mathit{self} \\
= & \{\text{inlining}\} \\
 & \lambda l'\ \rightarrow\ \textbf{let}\ \mathit{self}\ =\ re1\ \mathit{self}\ \mathbin{\rhd\!\!\!\|\!\!\lhd}\ (re2\ l')\ \textbf{in}\ \mathit{self}
\end{array}
$$

The reason for the different behaviour of the initial and the derived definition is that in the original definition self has type $(Regexp\ st)$, which is a functional, whereas in the derived definition, it has type $(Lexer\ s\ t)$, which is the datatype in which we want to create the cycle. Note in reference to the remark made at the end of Subsection 3.1 that this calculation depends on $\circledast$ being a binary operator (and not an unary postfix operator).

It remains the definition of $\oplus$, which is simple when expressed in terms of $\circledast$:

$$
\begin{array}{lll}
(\oplus) & :: & Regexp\ s\ t\ \rightarrow\ Regexp\ s\ t\ \rightarrow\ Regexp\ s\ t \\
re1 \oplus\ re2 & = & re1\ \rhd\ (re1 \circledast\ re2)
\end{array}
$$

4.4 Lexing

Given the state transition graph, we have to implement the lexing process by a traversal of the graphical DFA representation. Lazy evaluation of the graph ensures that only those portions of the table that are necessary for lexing the input at hand are actually evaluated and that each part of the graph is evaluated at most once. In Subsection 4.4, the definition of $execLexer$ used a function $lexOne$, which reads a single lexeme. In the following, we encode the graph traversal in $lexOne$ as a tail recursive function over the input string. The main complication in this function is the implementation of the principle of the longest match, i.e., we cannot take the first action that we encounter, but we have to take the last possible action before the automaton gets stuck. Therefore, during the traversal, we keep track of the last action that we passed along with the lexeme recognised up to this point. When no further transition is possible, we execute the last action on the associated lexeme; if there was no last action, we encountered a lexical error. The concrete definition of *lexOne* makes use of the recursive function *collect*, which collects all characters of the current lexeme. In addition to the current lexer and state, it maintains an accumulator for collecting the lexeme (third argument) and the most recently encountered action (fourth argument). The latter is initially an error, i.e., before the first valid action is encountered during lexing.

```
type OneToken s t = (Maybe t, Lexer s t, LexerState s)

lexOne        :: Lexer s t → LexerState s → OneToken s t
lexOne l state = collect l state "" (error "Lexical error!")
```

A value of type *OneToken s t* comprises all information associated with a lexeme: possibly a token and the lexer to be applied and state reached after reading the lexeme.

The implementation of *collect* distinguishes three cases, dependent on the next input symbol and the transition table *cls*:

```
collect :: Lexer s t → LexerState s t → String → OneToken s t
        → OneToken s t
collect (State a cls) state@(cs, pos, s) lexeme last =
  let last' = action a state lexeme last
  in
  case cs of
    []        → last'
    (c : cs') → case lookup c cls of
                  Nothing → last'
                  Just l'  → collect l' (cs', pos, s) (lexeme ++ [c]) last'
```

We use the auxiliary function *action* to compute the most recent action result *last'*. It applies actions to the current lexer state and the recognised lexeme; we shall return to its definition below. To decide how to react to the current input character *c*, we perform a *lookup* in the transition table *cls*.[4] If the character is not in the transition table (i.e., there is no valid transition in the DFA for the current input symbol), we use the most recent result of an action, namely *last'*. Otherwise, the automaton makes a transition to the state *l'*, which is associated with the current input symbol *c* in the transition table; furthermore, the current lexeme is extended with the character just read.

The function *action* deals with the three possible cases of the data type *LexAction* as follows:

```
action :: LexAction s t → LexerState s t → String → OneToken s t
       → OneToken s t
```

---

[4] The function `lookup :: Eq a ⇒ a → [(a, b)] → Maybe b` is from the Haskell prelude.

```
action NoAction _                     _        last = last
action (Action a) (cs, pos, s) lexeme _         =
  (a lexeme, l, (cs, advancePos pos lexeme, s))
action (Meta f)   (cs, pos, s) lexeme _         =
  let (pos', s', l') = f (advancePos pos lexeme) s
  in
  (Nothing, fromMaybe l l', (cs, pos', s'))
```

The function *advancePos* adjust the current position by the length of the given lexeme (its definition is not given, as we did not fix a concrete representation for positions). In the third equation of *action* (the case of a meta action), the function of the meta action is applied to yield a new position pos', user-defined state s', and possibly a new lexer l'. The function *fromMaybe* selects the new lexer if given; otherwise, it uses the current lexer l.[5] It should be clear that keeping track of the last action in the fourth argument of *collect* and *action* realises the principle of the longest match and, by means of lazy evaluation, delays the execution of the candidate actions until it is known which one is the last.

Accumulating the lexeme by appending individual list elements with $++$, as in the above definition of *collect* is inefficient. In the `Lexers` library, difference lists (similar to *ShowS* from the Haskell prelude) are used, which allow constant time appends.

5 Benchmarks

The performance of the presented technique was measured by coding a lexer for the programming language C with the `Lexers` library as well as implementing a *handcoded* lexer for the same set of regular expressions. The comparison between the two gives an estimate of the overhead incurred when using the more convenient `Lexers` library. The handcoded lexer implements the DFA using pattern matching on the first character of the remaining input. It carries the same overhead as `Lexers` for keeping track of token positions; there is still scope for optimisations, but they are not expected to make a big difference. The programs were compiled with the Glasgow Haskell Compiler (GHC), version 4.04, using optimisation level `-O`. The experiments were conducted on a 300Mhz Celeron processor under Linux. In the Haskell source of the library and the specification of the C lexer, no GHC-specific optimisations or language features were used; the input string was naively represented as a cons list of characters. The benchmark input to the lexer is (a) an artificial code, which contains a struct with 10 fields a 1000 times, and (b) the header file of the GTK+ GUI library after being expanded by a standard C pre-processor. Both files are about 200 kilobyte, i.e., they are real stress tests, not toy examples. The results are summarised in the following table (they do not include I/O time and are the best results from three runs on an unloaded machine):

		`Lexers`		handcoded		
input	size (byte)	time (sec)	tokens/sec	time (sec)	tokens/sec	%
GTK+	233,406	1.99	16,550	1.52	21,668	76%
structs	207,000	2.36	25,338	1.69	35,384	72%

---

[5] Haskell's standard library *Maybe* defines $fromMaybe :: a \rightarrow Maybe\ a \rightarrow a$.

These results show that for large input, the optimising lexer combinators are in the same ball park as a handcoded lexer, which needs about 70% of the execution time.

6 Related Work

In comparison to off-line scanner generators, the added flexibility and ease of use of the presented approach comes for increased startup costs (as the graph has to be build) and the inability to perform some optimisations on the automaton (like computing the minimal automaton). However, lazy evaluation avoids the startup costs partially if the given input requires only some of the regular expressions of the lexical specification.

Regarding related work, the main part of the paper made the relation to the automata theory underlying scanner generators clear. It would be interesting to establish a formal correspondence between the DFA created by the lexer combinators and those produced by the various off-line automata construction algorithms. I do not have a formal result yet, but believe that the automata is the same as the one constructed by the algorithm from the dragonbook [1] that constructs an DFA directly from a set of regular expression. Furthermore, the dragonbook [1, p. 128] mentions a technique called *lazy transition evaluation*. It also constructs a transition table at runtime, but the aim is to avoid the state explosion that can occur during the generation of a DFA from regular expressions—in the worst case, the number of states of the DFA can grow exponentially with the size of the regular expression. In this technique, state transitions are stored in a cache of fixed size. In case of a cache miss, the transition is computed using standard algorithms and stored in the cache; it may later be removed from the cache if it is not used for a while and many other transitions are needed. This technique, however, makes essential use of mutable data structures and is substantially different from the presented one, in both its aim and working principle—it was developed for the use in search routines of text editors.

Swierstra and Duponcheel's [7] parser combinators provided the inspiration for searching for a related technique that works for lexical analysis. Like their approach is based on the theory of SLL(1) stack automata, the present technique is based on the theory of finite state automata. The technique itself, however, is significantly different.

7 Conclusion

We have discussed an approach to lexical analysis in a lazy functional language that combines the ease of use and flexibility of combinator libraries with the efficiency of lexical analyser generators. In this approach, a lexical analyser is specified as a set of pairs of regular expressions and actions using functions from a combinator library. The combinators are smart constructors, building a state transition graph of a deterministic finite state automaton implementing the regular expressions. The graph is lazily constructed during lexing and makes use of cycles to represent the recursive combinators $\circledast$ and $\oplus$.

It is worth noting that the discussed technique makes essential use of laziness and can be regarded as a practical example of the usefulness of lazy evaluation. First of all, non-strictness allows to significantly optimise the implementation of recursive combinators by exploiting cyclic structures. Furthermore, the lazy construction of the state transition graph minimises startup costs when the input does not use all regular expressions of the lexical specification. However, the use of cyclic structures is definitely an advanced feature of Haskell, which requires some experience to use; we showed how equational reasoning can help to compute an efficient solution that makes use of cyclic structures from a less efficient, but easier to understand implementation. Furthermore, the implementation of the principle of the longest match is simplified by using lazy evaluation to keep track of the last action.

The use of a lazy list of characters for the input to the lexer is definitely not the most efficient choice. However, it simplifies the presentation and only the function *collect* depends on the representation of the input. It should not be hard to adapt it to a more efficient form of reading the character stream. Furthermore, it might seem attractive to represent lexers as monads; this presents, however, some technical challenges, similar to those reported in [7].

Regarding future work, it would be interesting to investigate whether the whole algorithm could be formally derived from the underlying automata theory, which was only informally described in this paper.

Acknowledgements. I am indebted to Simon Peyton Jones for a number of suggestions that helped to improve this paper and the `Lexers` library considerably. Furthermore, I am grateful to Gabriele Keller and the anonymous referees for their helpful comments and suggestions on the paper. Finally, I like to thank Roman Lechtchinsky for his feedback on the `Lexers` library.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
2. Andrew W. Appel, James S. Mattson, and David R. Tardit. A lexical analyzer generator for Standard ML. `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Lex/manual.html`, 1994.
3. Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.
4. Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.
5. Simon Marlow. Happy user guide. `http://www.dcs.gla.ac.uk/fp/software/happy/doc/happy.html`, 1997.
6. Rishiyur S. Nikhil, Arvind, James E. Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science, 1995.
7. S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.
8. David R. Tarditi and Andrew W. Appel. ML-Yacc user's manual. `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/manual.html`, 1994.
9. William W. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, second edition, 1985.